

Backtracking

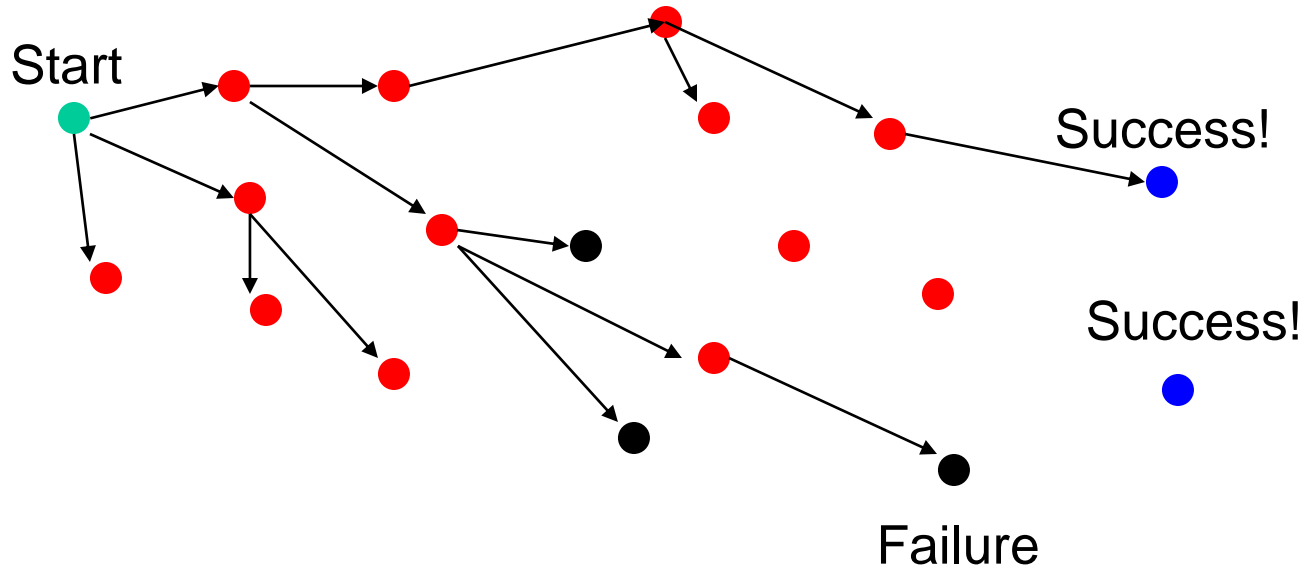
Bruce Nan



Backtracking

- Suppose you have to make a series of *decisions*, among various *choices*, where
 - You don't have enough information to know what to choose
 - Each decision leads to a new set of choices
 - Some sequence of choices (possibly more than one) may be a solution to your problem
- **Backtracking** is a methodical way of trying out various sequences of decisions, until you find one that “works”

Backtracking



Problem space consists of states (nodes) and actions (paths that lead to new states). When in a node can only see paths to connected nodes

If a node only leads to failure go back to its "parent" node. Try other alternatives. If these all lead to failure then more backtracking may be necessary.

A More Concrete Example

- ▶ Sudoku
- ▶ 9 by 9 matrix with some numbers filled in
- ▶ all numbers must be between 1 and 9
- ▶ Goal: Each row, each column, and each mini matrix must contain the numbers between 1 and 9 once each
 - no duplicates in rows, columns, or mini matrices

5	3			7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

Solving Sudoku – Brute Force

- ▶ A brute force algorithm is a simple but general approach
- ▶ Try all combinations until you find one that works
- ▶ This approach isn't clever, but computers are fast
- ▶ Then try and improve on the brute force results

5	3			7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

Solving Sudoku

- ▶ Brute force Sudoku Solution
 - if not open cells, solved
 - scan cells from left to right, top to bottom for first open cell
 - When an open cell is found start cycling through digits 1 to 9.
 - When a digit is placed check that the set up is legal
 - now solve the board

5	3	1		7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

Question 1

► After placing a number in a cell is the remaining problem very similar to the original problem?

A. Yes

B. No

Solving Sudoku – Later Steps

5	3	1		7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9



5	3	1	2	7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9



5	3	1	2	7	4			
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

5	3	1	2	7	4	8		
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

5	3	1	2	7	4	8	9	
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

uh oh!

Sudoku – A Dead End

- ▶ We have reached a dead end in our search

5	3	1	2	7	4	8	9	
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

- ▶ With the current set up none of the nine digits work in the top right corner

Backing Up

- ▶ When the search reaches a dead end in **backs up** to the previous cell it was trying to fill and goes onto to the next digit
- ▶ We would back up to the cell with a 9 and that turns out to be a dead end as well so we back up again
 - so the algorithm needs to remember what digit to try next
- ▶ Now in the cell with the 8. We try and 9 and move forward again.

5	3	1	2	7	4	8	9	
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

5	3	1	2	7	4	9		
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

Characteristics of Brute Force and Backtracking

- ▶ Brute force algorithms are slow
- ▶ They don't employ a lot of logic
 - For example we know a 6 can't go in the last 3 columns of the first row, but the brute force algorithm will plow ahead any way
- ▶ But, brute force algorithms are fairly easy to implement as a first pass solution
 - backtracking is a form of a brute force algorithm

Key Insights

- ▶ After trying placing a digit in a cell we want to solve the new sudoku board
 - Isn't that a smaller (or simpler version) of the same problem we started with?!?!?!?
- ▶ After placing a number in a cell the we need to remember the next number to try in case things don't work out.
- ▶ We need to know if things worked out (found a solution) or they didn't, and if they didn't try the next number
- ▶ If we try all numbers and none of them work in our cell we need to *report back* that things didn't work

Recursive Backtracking

- ▶ Problems such as Suduko can be solved using recursive backtracking
- ▶ recursive because later versions of the problem are just slightly simpler versions of the original
- ▶ backtracking because we may have to try different alternatives

Recursive Backtracking

Pseudo code for recursive backtracking algorithms

If at a solution, report success

for(every possible choice from current state / node)

Make that choice and take one step along path

Use recursion to solve the problem for the new node / state

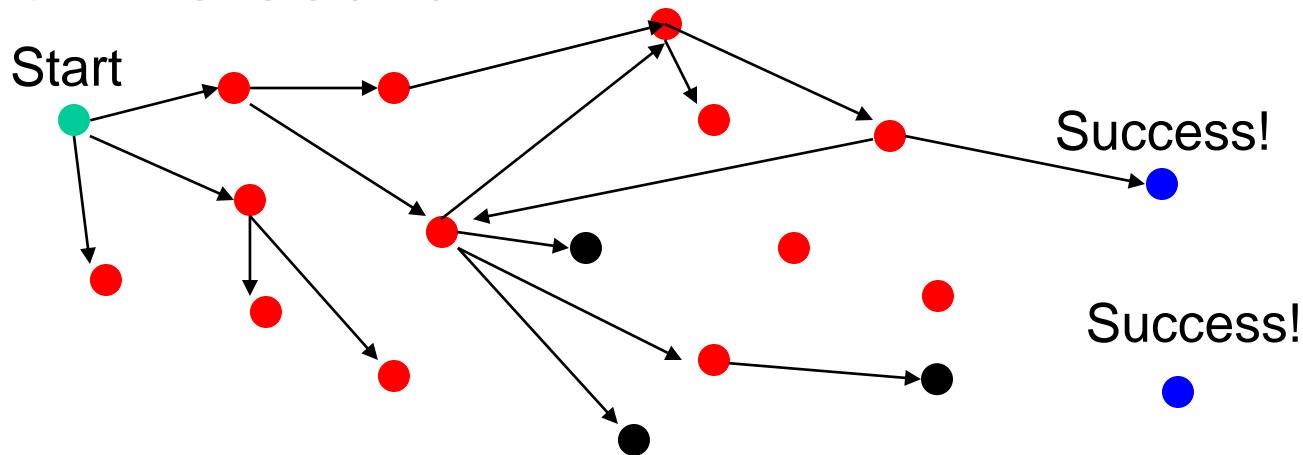
If the recursive call succeeds, report the success to the next high level

Back out of the current choice to restore the state at the beginning of the loop.

Report failure

Goals of Backtracking

- ▶ Possible goals
 - Find a path to success
 - Find all paths to success
 - Find the best path to success
- ▶ Not all problems are exactly alike, and finding one success node may not be the end of the search



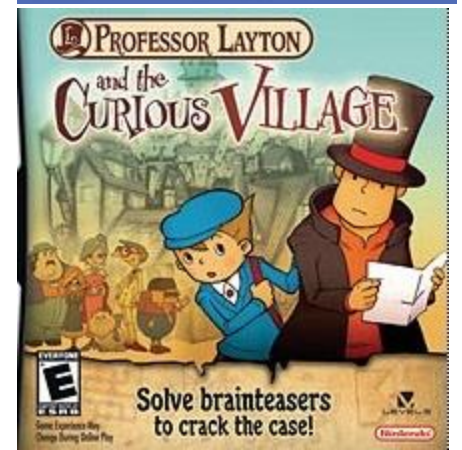
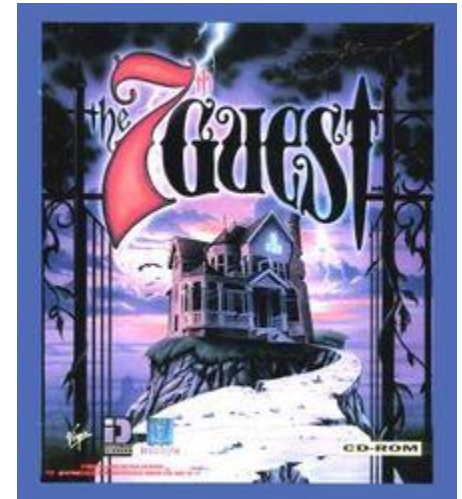
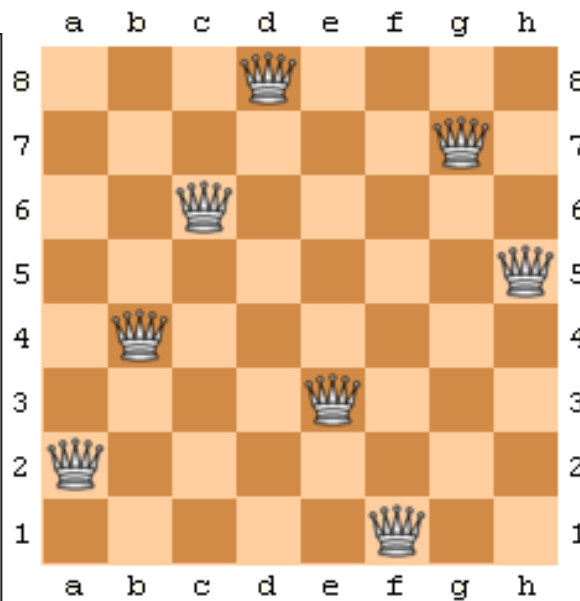
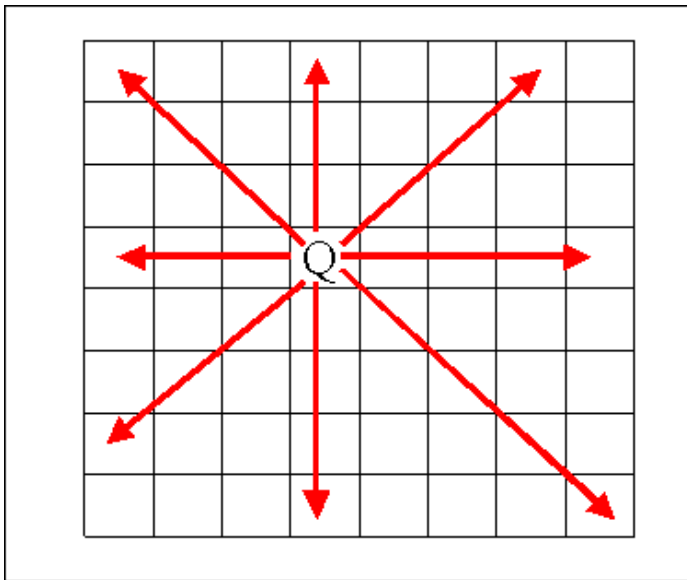


The 8 Queens Problem



The 8 Queens Problem

- ▶ A classic chess puzzle
 - Place 8 queen pieces on a chess board so that none of them can attack one another



The N Queens Problem

- ▶ Place N Queens on an N by N chessboard so that none of them can attack each other
- ▶ Number of possible placements?
- ▶ In 8 x 8

$$\begin{aligned} & 64 * 63 * 62 * 61 * 60 * 59 * 58 * 57 \\ & = 178,462, 987, 637, 760 / 8! \\ & = 4,426,165.368 \end{aligned}$$

$$\binom{n}{k} = \frac{n \cdot (n-1) \cdots (n-k+1)}{k \cdot (k-1) \cdots 1} = \frac{n!}{k!(n-k)!} \quad \text{if } 0 \leq k \leq n \quad (1)$$

n choose k

- How many ways can you choose k things from a set of n items?
- In this case there are 64 squares and we want to choose 8 of them to put queens on

Question 2

► For valid solutions how many queens can be placed in a give column?

A. 0

B. 1

C. 2

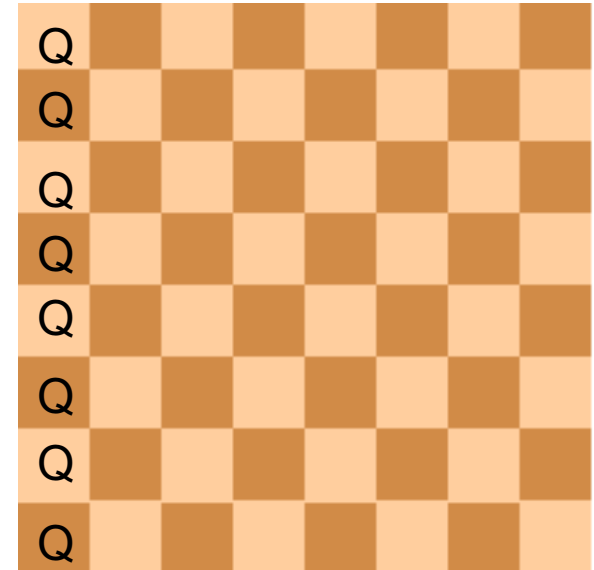
D. 3

E. 4

F. Any number

Reducing the Search Space

- ▶ The previous calculation includes set ups like this one
- ▶ Includes lots of set ups with multiple queens in the same column
- ▶ How many queens can there be in one column?
- ▶ Number of set ups
 $8 * 8 * 8 * 8 * 8 * 8 * 8 * 8 = 16,777,216$
- ▶ We have reduced search space by two orders of magnitude by applying some logic



A Solution to 8 Queens

- If number of queens is fixed and I realize there can't be more than one queen per column I can iterate through the rows for each column

```
for(int c0 = 0; c0 < 8; c0++){  
    board[c0][0] = 'q';  
    for(int c1 = 0; c1 < 8; c1++){  
        board[c1][1] = 'q';  
        for(int c2 = 0; c2 < 8; c2++){  
            board[c2][2] = 'q';  
            // a little later  
            for(int c7 = 0; c7 < 8; c7++){  
                board[c7][7] = 'q';  
                if( queensAreSafe(board) )  
                    printSolution(board);  
                board[c7][7] = ' '; //pick up queen  
            }  
            board[c6][6] = ' '; // pick up queen
```

Eight Queen Problem: Algorithm

```
putQueen(row)
{
    for every position col on the same row
        if position col is available
            place the next queen in position col
        if (row<8)
            putQueen(row+1);
        else success;
        remove the queen from position col
}
```

Eight Queen Problem: Implementation

- Define an 8 by 8 array of 1s and 0s to represent the chessboard
- The array is initialized to 1s, and when a queen is put in a position (c,r), board[r][c] is set to zero
- Note that the search space is very huge:
16,772, 216 possibilities.
- Is there a way to reduce search space?
Yes Search Pruning.

Eight Queen Problem: Implementation

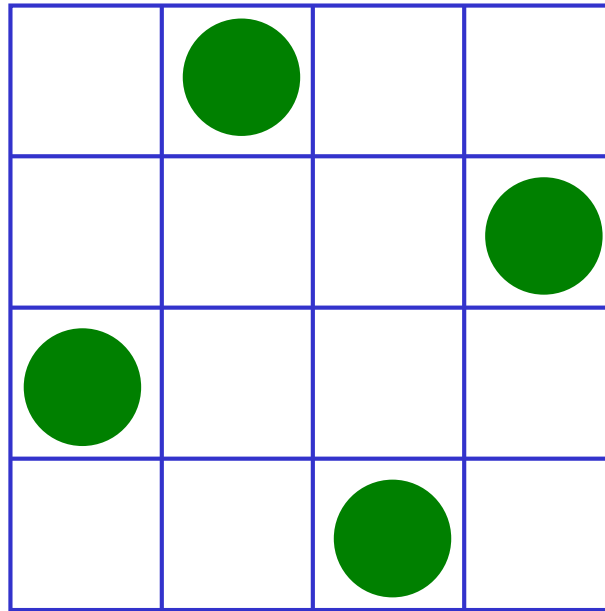
- ▶ We know that for queens:
 - each row will have exactly one queen
 - each column will have exactly one queen
 - each diagonal will have at most one queen
- ▶ This will help us to model the chessboard not as a 2-D array, but as a set of rows, columns and diagonals.
- ▶ To simplify the presentation, we will study for smaller chessboard, 4 by 4

Implementing the Chessboard

- First: we need to define an array to store the location of so far placed queens

PositionInRow



1
3
0
2



Implementing the Chessboard Cont'd

- ▶ We need an array to keep track of the availability status of the column when we assign queens.

Suppose that we
have placed two
queens

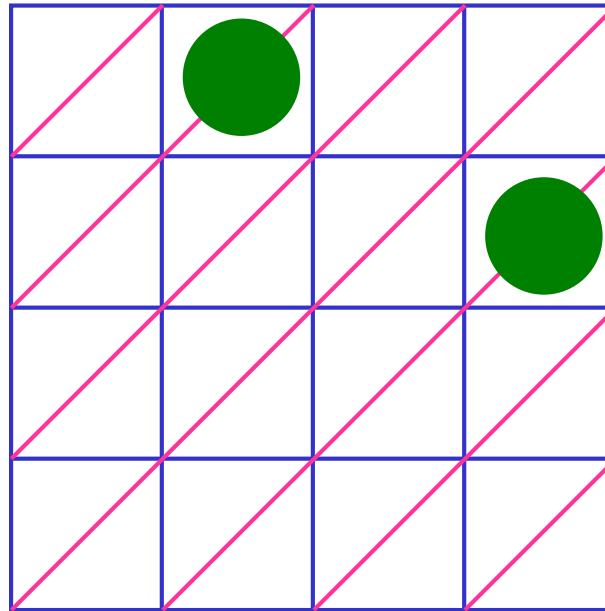
			
			

T	F	T	F
---	---	---	---

Implementing the Chessboard Cont'd

- ▶ We have 7 left diagonals, we want to keep track of available diagonals after the so far allocated queens

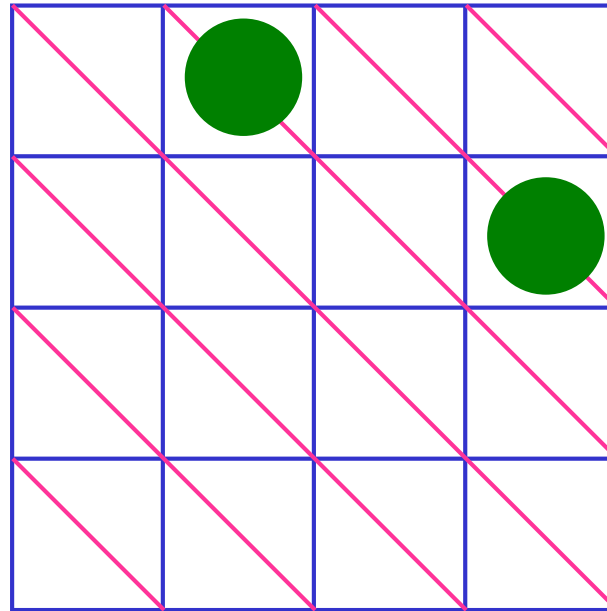
T
F
T
T
F
T
T



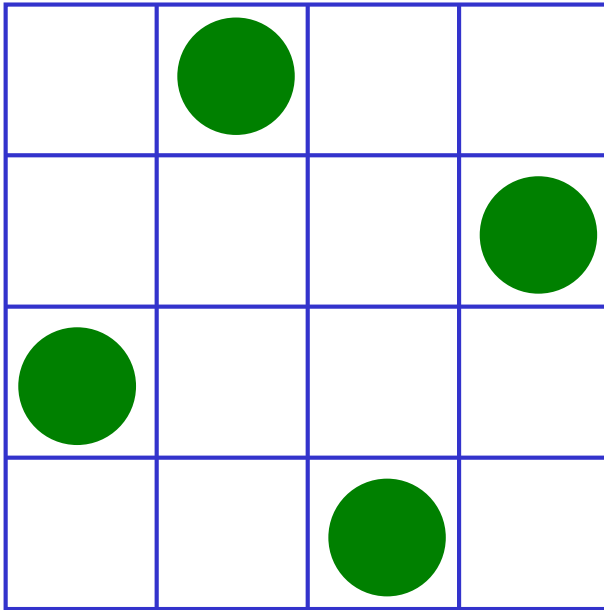
Implementing the Chessboard Cont'd

- ▶ We have 7 left diagonals, we want to keep track of available diagonals after the so far allocated queens

T
F
F
T
T
T
T



Backtracking



[4,3]
[3,1]
[2,4]
[1,2]

The putQueen Recursive Method

```
static void putQueen(int row){
    for (int col=0;col<squares;col++)
        if (column[col]==available && leftDiagonal[row+col]==available &&
            rightDiagonal[row-col+norm]== available)
        {
            positionInRow[row]=col;
            column[col]=!available;
            leftDiagonal[row+col]=!available;
            rightDiagonal[row-col+norm]=!available;
            if (row< squares-1)
                putQueen(row+1);
            else
                System.out.println(" solution found");
            column[col]=available;
            leftDiagonal[row+col]=available;
            rightDiagonal[row-col+norm]= available;
        }
}
```

N Queens

- ▶ The *problem* with N queens is you don't know how many for loops to write.
- ▶ Do the problem recursively
- ▶ Write recursive code with class and demo
 - show backtracking with breakpoint and debugging option

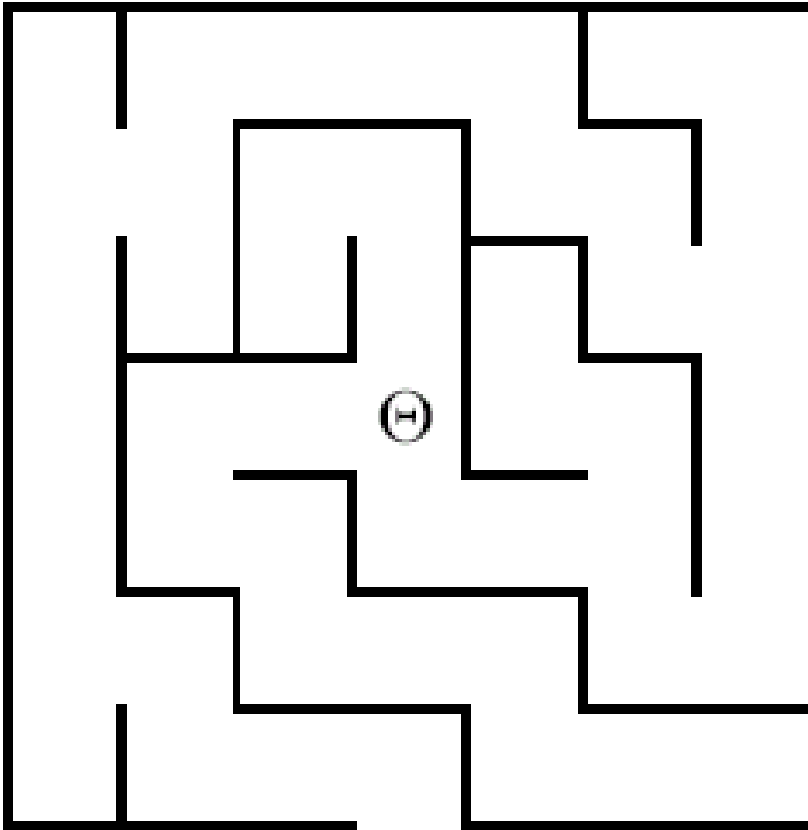
Recursive Backtracking

- ▶ You must practice!!!
- ▶ Learn to recognize problems that fit the pattern
- ▶ Is a *kickoff* method needed?
- ▶ All solutions or a solution?
- ▶ Reporting results and acting on results

THANKS

Another Backtracking Problem

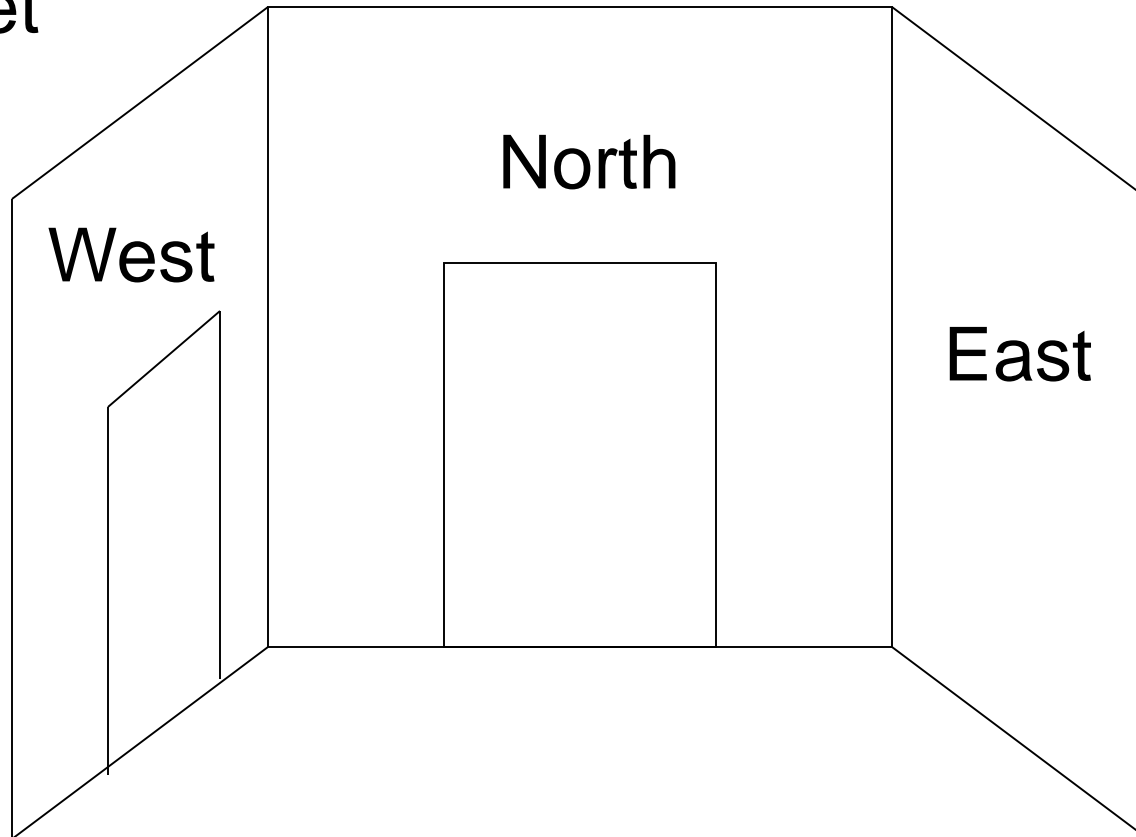
A Simple Maze



Search maze until way out is found. If no way out possible report that.

The Local View

Which way do
I go to get
out?

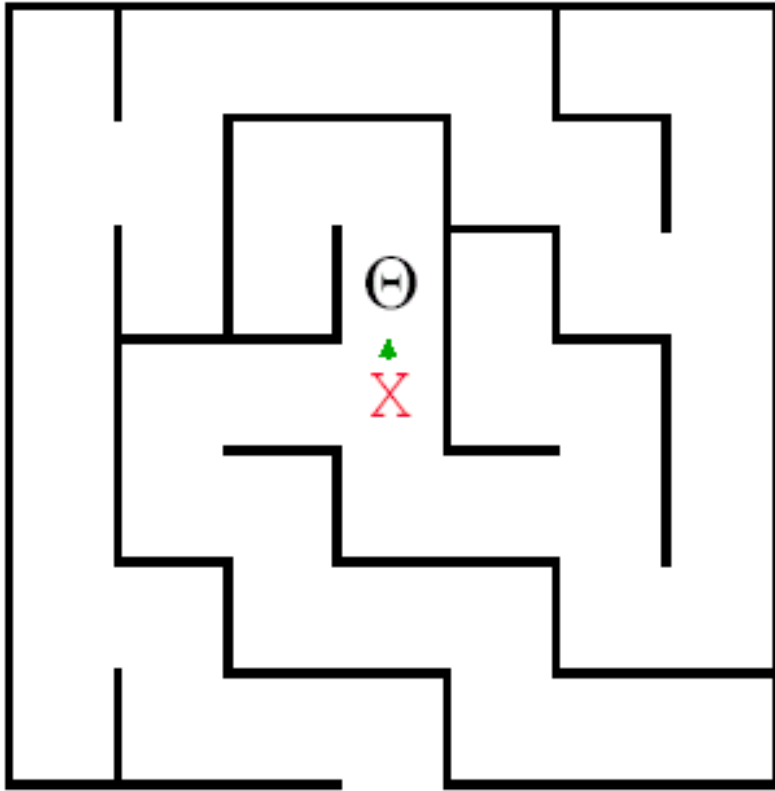


Behind me, to the South
is a door leading South

Modified Backtracking Algorithm for Maze

- ▶ If the current square is outside, return TRUE to indicate that a solution has been found.
- If the current square is marked, return FALSE to indicate that this path has been tried.
- Mark the current square.
- for (each of the four compass directions)
 - { if (this direction is not blocked by a wall)
 - { Move one step in the indicated direction from the current square.
 - Try to solve the maze from there by making a recursive call.
 - If this call shows the maze to be solvable, return TRUE to indicate that fact.
 - }
- }
- Unmark the current square.
- Return FALSE to indicate that none of the four directions led to a solution.

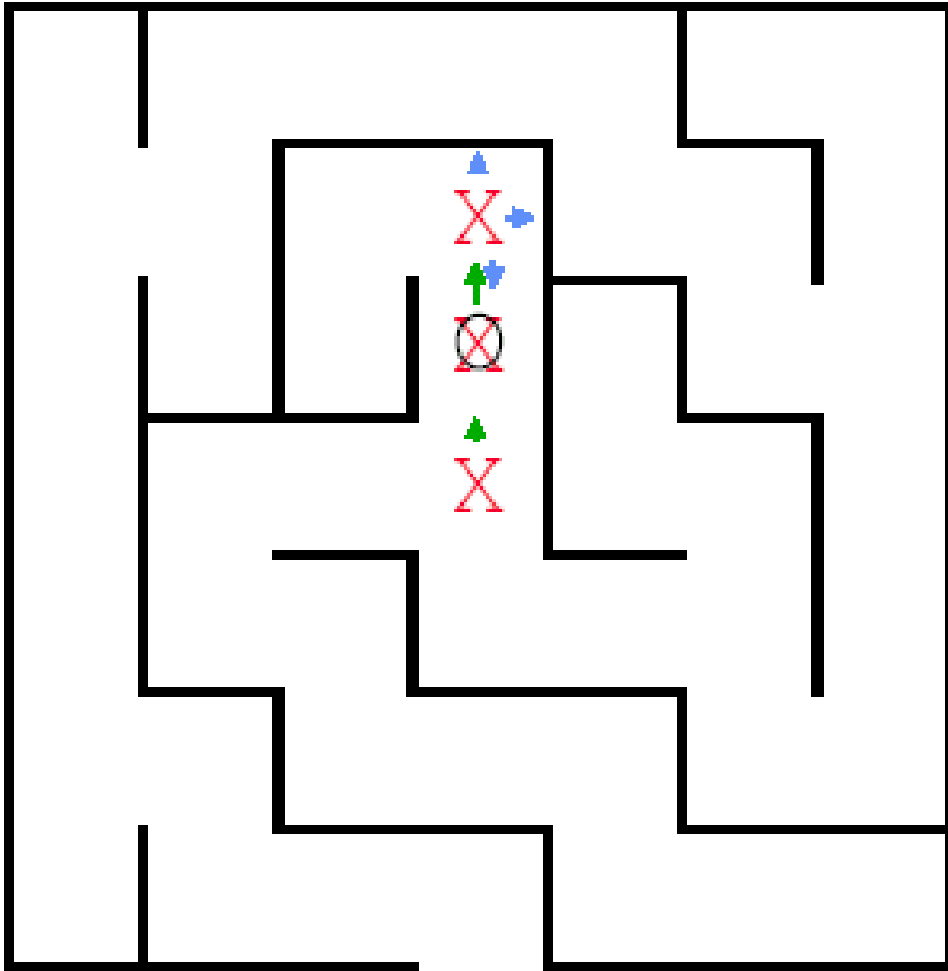
Backtracking in Action



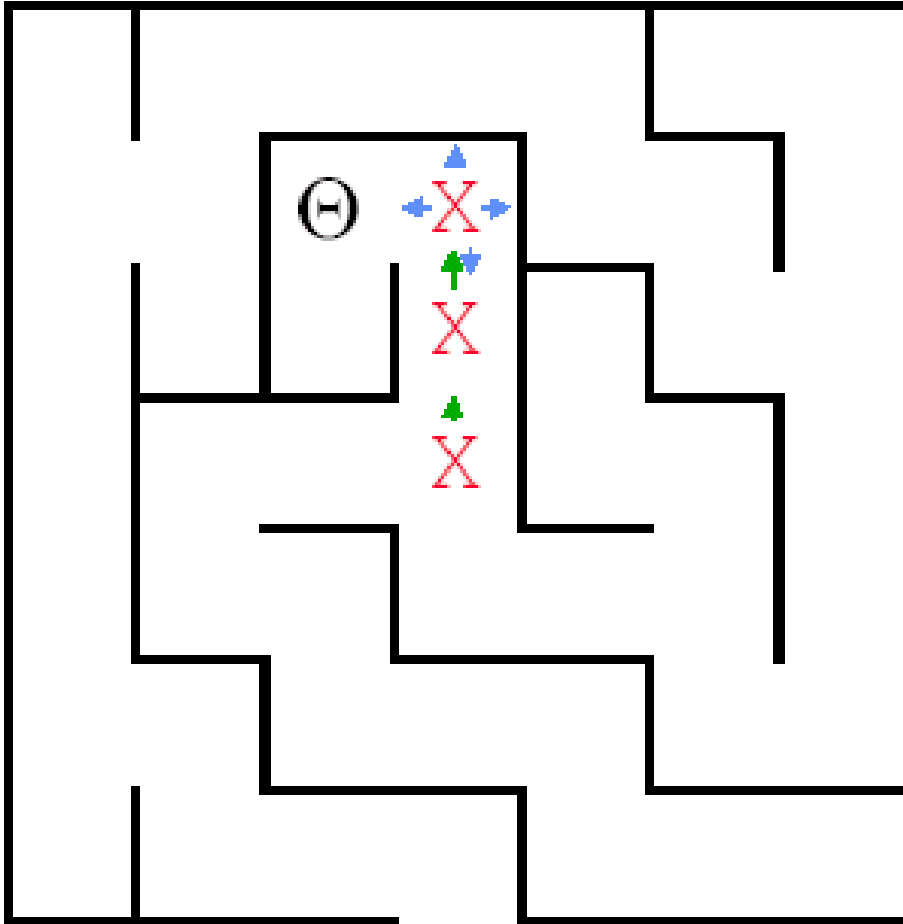
The crucial part of the algorithm is the for loop that takes us through the alternatives from the current square. Here we have moved to the North.

```
for (dir = North; dir <= West; dir++)  
{  
    if (!WallExists(pt, dir))  
        {if (SolveMaze(AdjacentPoint(pt, dir)))  
            return(TRUE) ;  
        }  
}
```

Backtracking in Action

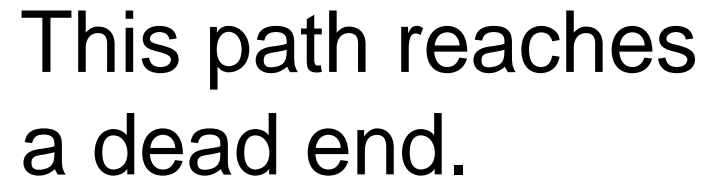


Here we have moved North again, but there is a wall to the North . East is also blocked, so we try South. That call discovers that the square is marked, so it just returns.

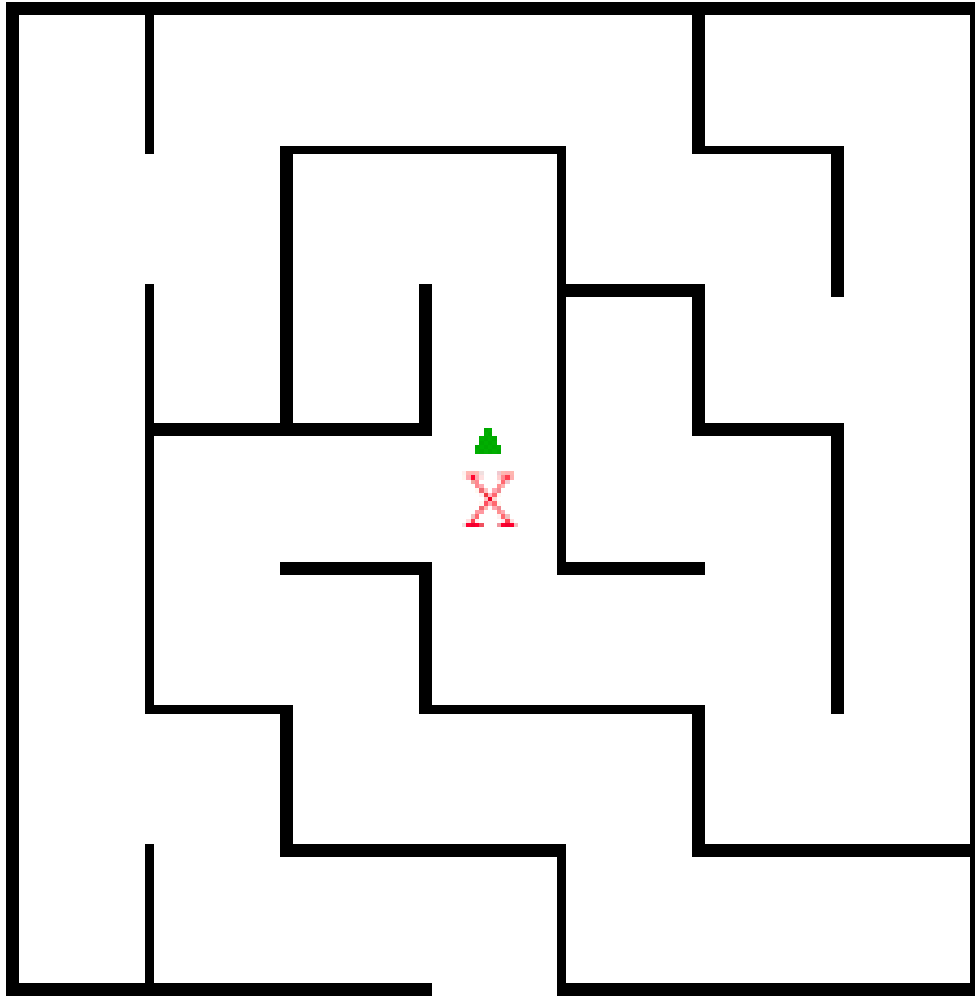


So the next move we can make is West.

Where is this leading?

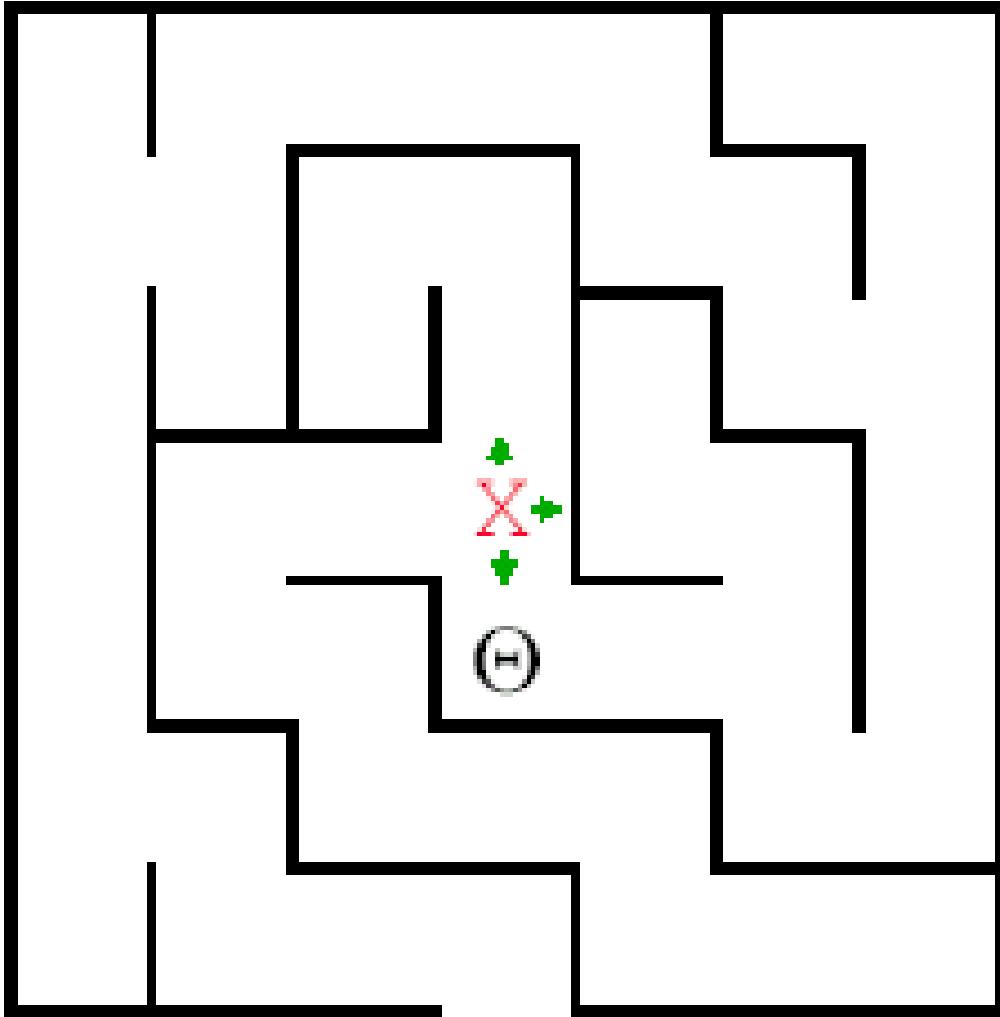


Remember the program stack!



The recursive calls
end and return until
we find
ourselves back here.

And now we try
South



Path Eventually Found

